File Management

CS3026 Operating Systems

Lecture 20

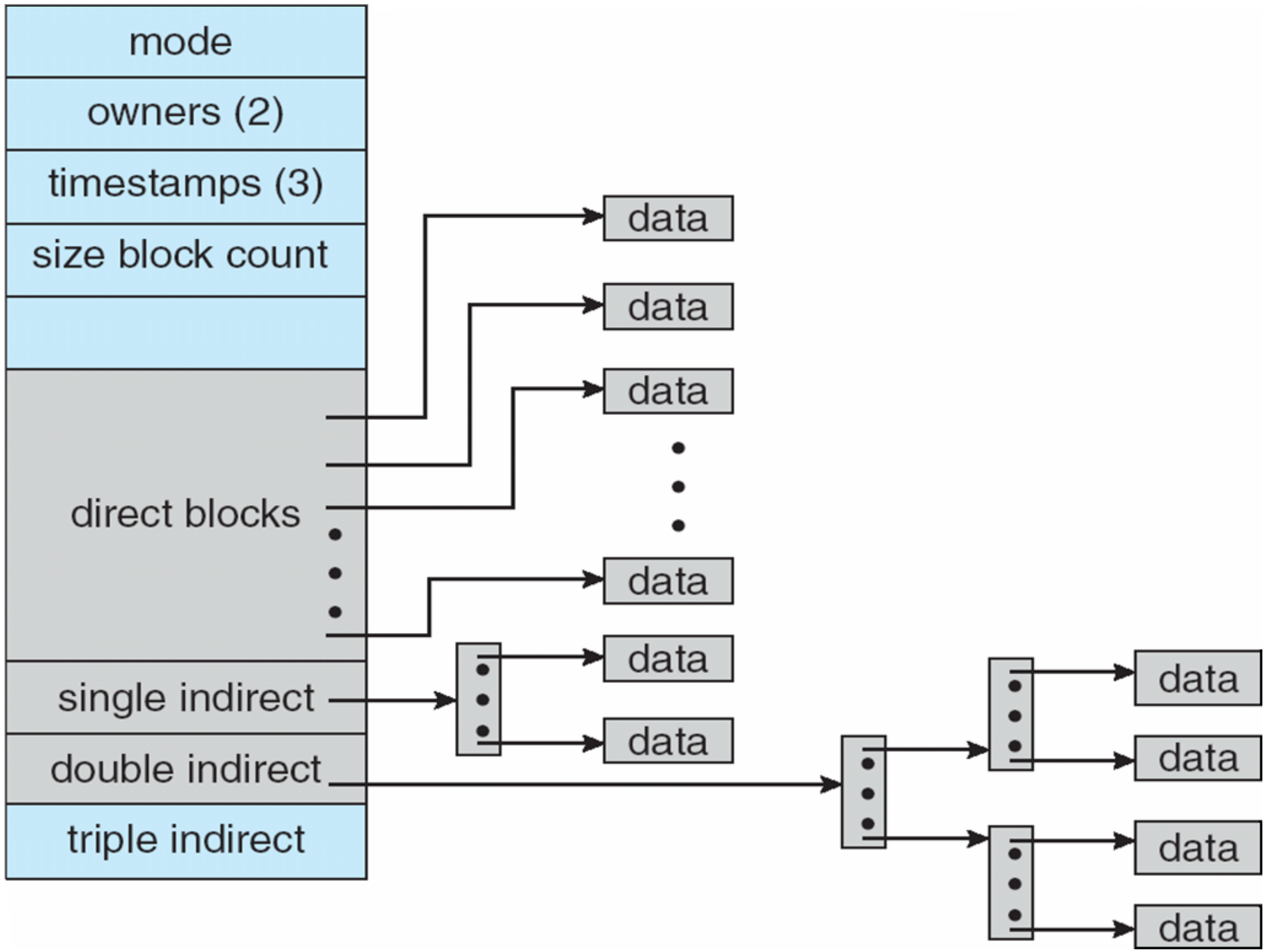
Indexed Allocation: i-Nodes

- i-Node manages n-level index
  - Entry in the i-Node points to a block on disk that contains pointers to other blocks
- How can we distinguish between index blocks and data blocks?
- How do we know how many levels the index has?

Direct and Indirect Referencing

- First N index entries point directly to the first N blocks allocated for the file
- If file is longer than N blocks, more levels of indirection are used
- Inode contains three index entries for “indirect” addressing
  - “single indirect” address:
    - Points to an intermediate block containing a list of pointers
  - “double indirect” address:
    - Points to two levels of intermediate pointer lists
  - “triple indirect” address:
    - Points to three levels of intermediate pointer lists

i-Node Indexed References of Disk Blocks



i-Node Direct and Indirect Indexing

- Example implementation with 13 index entries:
  - i-Node contains a list of 13 index entries that combine four different forms of index
  - Direct block references:
    - 10 entries of this list point directly to file data blocks
  - Single indirect (two levels):
    - Entry 11 is regarded as always pointing to an index disk block: this index block contains address of actual file data blocks
  - Double indirect (three levels): entry 12 is regarded to be the starting point of a three-level index
  - Triple indirect (four levels): entry 13 is regarded to be the starting point of a four-level index

i-Node Direct and Indirect Indexing

- Based on which entry in the i-Node is used, the file system management can distinguish whether an indexed block is a data block or another level of one of the indices
- Assumption
  - There are many small files, the number of directly referenced blocks may be enough
  - For larger files, the additional indices are used

i-Node Table

- Operating system has to manage the i-Node table
  - When a file is opened / created, its i-Node is loaded into the i-Node table
  - The size of this table determines the number of file that can be held open at the same time

File Allocation with i-Nodes

- What is maximum size of a file that can be indexed:
  - Depends of the capacity of a fixed-sized block
- Example implementation with 15 index entries:
  - 12 direct, single (13) / double (14) / triple (15) indirect
  - Block size 4kb, holds 512 block addresses (32-bit addresses)

Level	Number of Blocks	Number of Bytes
Direct	12	48K
Single Indirect	512	2M
Double Indirect	$512 \times 512 = 256K$	1G
Triple Indirect	$512 \times 256K = 128M$	512G

i-Nodes

- Advantage
  - i-Node is only loaded into memory when a file is opened
  - Good for managing very large disks efficiently
  - We need a list of i-Nodes of open files: size of this list determines how many files may be open at the same time
- Disadvantage
  - The i-Node only has a fixed list for block references
  - If a file is small, fast and efficient management
  - If file is large, the i-Node has to be extended with a hierarchy of indirect block lists connected to the i-Node, needs extra I/O operations to scan the index

Free Space Management

- Just as allocated space must be managed, so must unallocated space
- It is necessary to know which blocks are available
- Methods
  - Bit tables: for each block one bit (used, unused)
    - As small as possible
  - Free portions chained together
    - Each time a block is allocated, it has to be read first get the pointer to the next free block
  - Indexing
    - Treats free space as a file
    - Create pool of free i-nodes and free disk blocks

Free Space Management

- Bit Table
  - Vector of bits: each bit for one disk block
  - Is as small as possible
- Can still be of considerable size:
  - Amount of memory (bytes) needed:

  Disk size in bytes / 8 x file system block size
- Example:
  - 16 GB hard disk, block size 512 bytes: bit table occupies 4 MB, requires 8000 disk blocks when stored on the disk

Free Space Management

- Chained Free blocks
  - We can chain free blocks together
  - Each free block contains a pointer to next free block
- Problem
  - Disk may become fragmented
  - When a free block is allocated, it has to be read from disk first to retrieve the “next free block pointer”
  - Creation / deletion of files may become slow over time

Free Space Management

- Indexing:
  - Free space is treated like a file collecting all the free blocks
- Free Block List:
  - Each block is assigned a number sequentially
  - The list of numbers of all free blocks is maintained in a reserved portion of the disk

Directories

- Directories maintain information about files
  - File name
  - Location of actual data related to such a file name
- File name is a symbolic representation of data stored on disk
- Directory entry
  - File name
  - File attributes
  - Physical address of the file data
- Directory structure
  - Simple list
  - Hierarchical, tree structure: directories contain sub-directories

Hierarchical Directories

- Unix uses a hierarchy of directories
- Top-level directory: root
  - All other directories are sub-directories of root
- Path:
  - Is the sequence of subdirectories to reach a file
- Path name:
  - Absolute: uniquely identifies a file within the directory hierarchy
    - Starts with root
    - Example: "/usr/local/myname/myfile.txt"
  - Relative: identifies a file, starting from the current working directory
    - Example:
      - working directory: "/usr"
      - Path name: "local/myname/myfile.txt"
- Special files in a directory:
  - "." points to the directory itself: "./myfile.txt"
  - ".." points to the parent directory: "../myname/myfile.txt"

Directories in Unix

- Structured as a tree
  - Each directory contains files and/or other sub-directories
- Implementation:
  - A directory is a file that contains a list of file names and a reference to the corresponding inode in the inode table of a volume
  - Inode reference:
    - Is the so-called “i-number”: index into the inode table

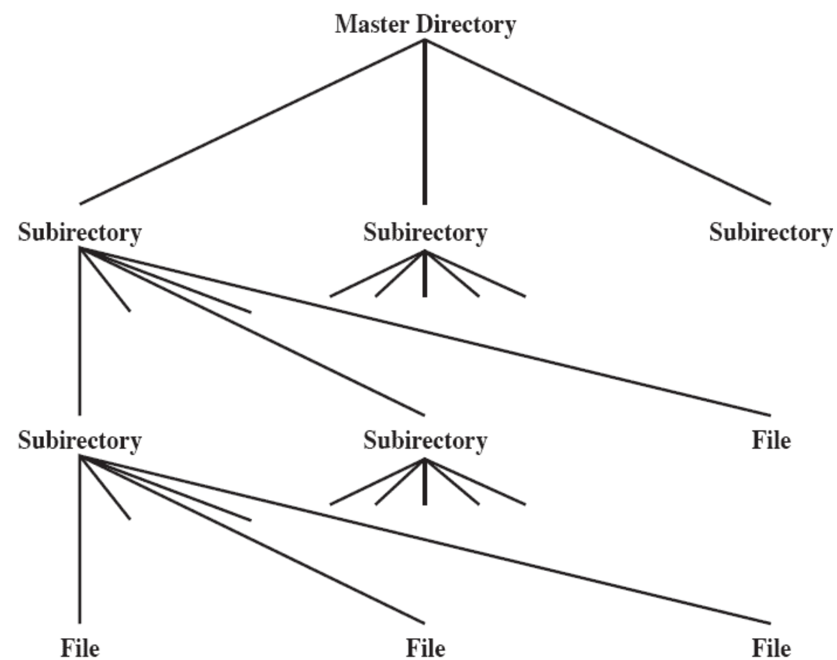


Figure 12.4 Tree-Structured Directory

Unix Directories and i-Nodes

- Directories are structured as a tree
- Directory entries contain filename and associated i-number
- Is the index into the i-Node table

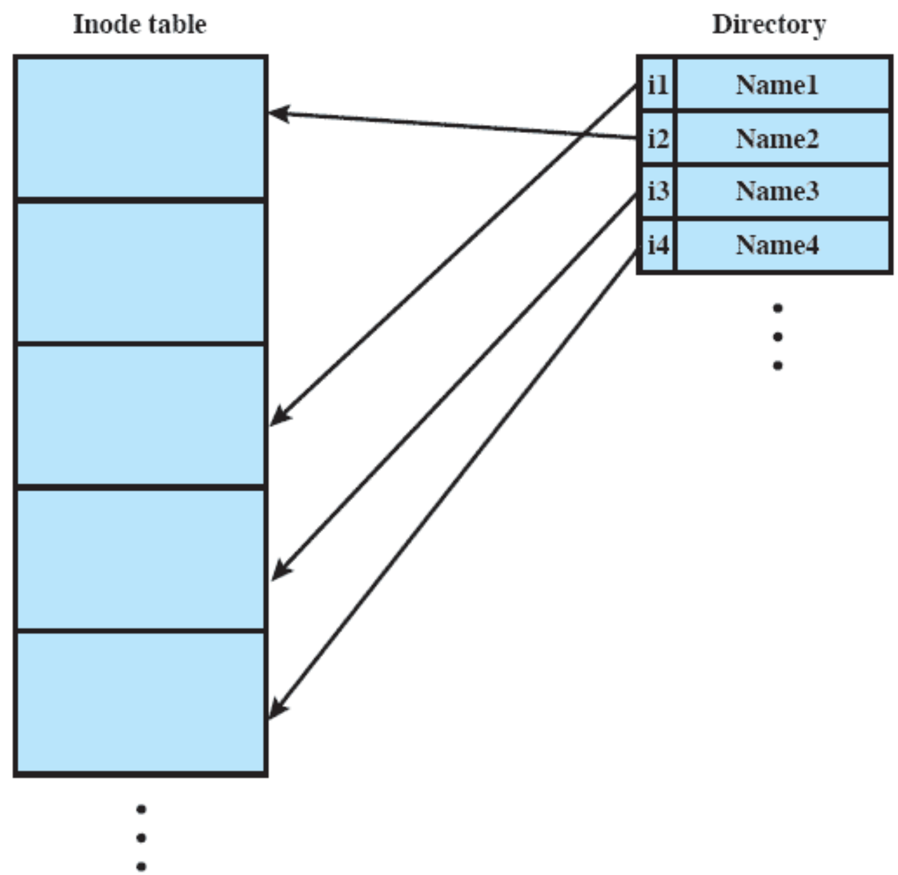


Figure 12.15 UNIX Directories and Inodes

Recovery

Write Operations

- Creating a new file / deleting files results in a set of necessary write operations
  - Write meta-data:
    - Write out a changed directory i-node and disk block containing the directory info
    - Write out a new i-node for the file itself
  - Write out data blocks for file

File System Performance

- Is achieved with caches
- Buffer cache
  - Hold data in memory, perform read / write operation much faster
  - Needs some form of block management
    - When a disk block is updated, it must be found in cache, use of cache data structures that support this search
  - Is a danger to file system integrity
  - Unix: system call "sync()" that allows to force a write of cache content
- Write-through cache
  - Disk access for each write operation, data is kept in cache for fast read
  - More secure, less performance

Failure Situations

- Design Considerations:
  - Direct access to hard disk extremely slow compared to memory access
  - Therefore: use of disk caches
    - Many read / write activities can directly be done on the cache and slow disk access avoided
  - Data held in memory
    - Directory information, Free block list, i-Node table
- Cached systems leave data in a precarious state
  - Structural changes not written back to disk
    - E.g.: creating new file or deleting files
  - Exposed to danger of system failure, loss of data

Recover from System Failure

- File system must be able to detect problem and correct them
  - File system repair:
    - check to detect inconsistencies and repair them
    - Unix command: fsck
  - File system robustness:
    - make file system robust against failure with transaction concepts
    - Journaling File Systems
    - Log-structured file systems

System Failure

- Example: delete a file
  - Remove file from its directory
  - Release i-Node of file to pool of free i-Nodes
  - Put all the disk blocks of the file into the pool of free disk blocks
- System crash after first step
  - Reference to i-Node deleted, no other reference from free pool established
    - i-Node and all file disk blocks are orphaned and cannot be re-allocated

Journaling File Systems

- Transaction-oriented (journaling) file system
  - File system implementation is inspired by log-based recovery algorithms for database systems
- Transaction-oriented manipulation of file system data
  - Each write operation occurs as a transaction (atomic action)
  - Transactions are logged
  - Log-based recovery in case of disruptions

Journaling File System

- Idea
  - Keep a small circular log on disk
  - Record in this what write actions will be performed on the file system, before actual disk operations occur
- Log file helps to recover from system crashes
  - After a system crash, operating system can take information from log file and perform a “roll forward”:
    - Finish write operations as recorded in journal

Journaling File System

- Manipulation done in the form of transactions
  - First, recording of operations in log (begin of transaction)
    - E.g.: All three actions necessary for deleting a file are recorded
    - Log entry is written to disk
  - Second, after log recordings, actual write operations begin
  - Third, when all write operation on disk successful (across system crashes), log entry is deleted (end of transaction)

Journaling File System

- Normal behaviour
  - All actions recorded in the log have to happen eventually
  - Execute operations held in log
  - When finished executed (all data on disk), remove entry from log
- Recovery after system crash: “Roll forward”
  - All actions recorded in the log have to happen eventually
  - With the log, these actions can be “re-played” and manipulations completed
  - Same as “normal behaviour”: operations in log are executed and removed from log after completion

Journaling File Systems

- Robust in case of system crashes
  - Logs are checked after recovery and logged actions redone
  - this can be done multiple times if there are multiple system crashes
- Changes to file system are atomic
- Logged operations must be “idempotent”
  - Can be repeated as often without harm
  - E.g.: “mark i-node *k* as free” can be done over and over again

Journaling File System

- Physical journals
  - Logs an advance copy of each disk block later to be written to the main file system
  - If there is a crash, write can be replayed from journal
- What if there is a crash during write of journal
  - We need info that journal entry is a complete and valid
    - Store checksum: if checksum of entry does not match stored checksum, we can ignore this entry
- Physical journal have a performance problem
  - Each changed block is written twice
- Provide good fault protection

Journaling File System

- Logical journals
  - Stores only changed metadata
  - Less fault tolerant, better performance
  - Recovers quickly after crash, but there is the danger that unjournaled file data and journaled meta data fall out of sync
- Example: extending a file with additional data
  - Three separate writes
    - Record the additional reference to data in i-node, point to new data blocks
    - Change the free-space list, re-allocate the blocks
    - Actually write the data blocks out to disk
  - Only the metadata will be logged in journal, the actual write to disk of the data is not logged
  - After recovery, write of metadata is replayed and done, but the content of the disk blocks is lost

Log Structured File Systems

- Basic idea:
  - Structure the entire disk as a huge circular log file
  - All updates to data / metadata (i-nodes) are written sequentially to a continuous stream, called a “log”
    - Write operations are first buffered in memory
    - Periodically, write a whole segment of these operations out to disk and add it to the head of the log file
    - Newest updates always at the beginning of the log file, overwriting its own tail
    - Hold multiple versions of a data object in chronological order in this log
  - Designed for high write throughput
  - Hold data in cache for fast read operations
    - No need to search for data on disk

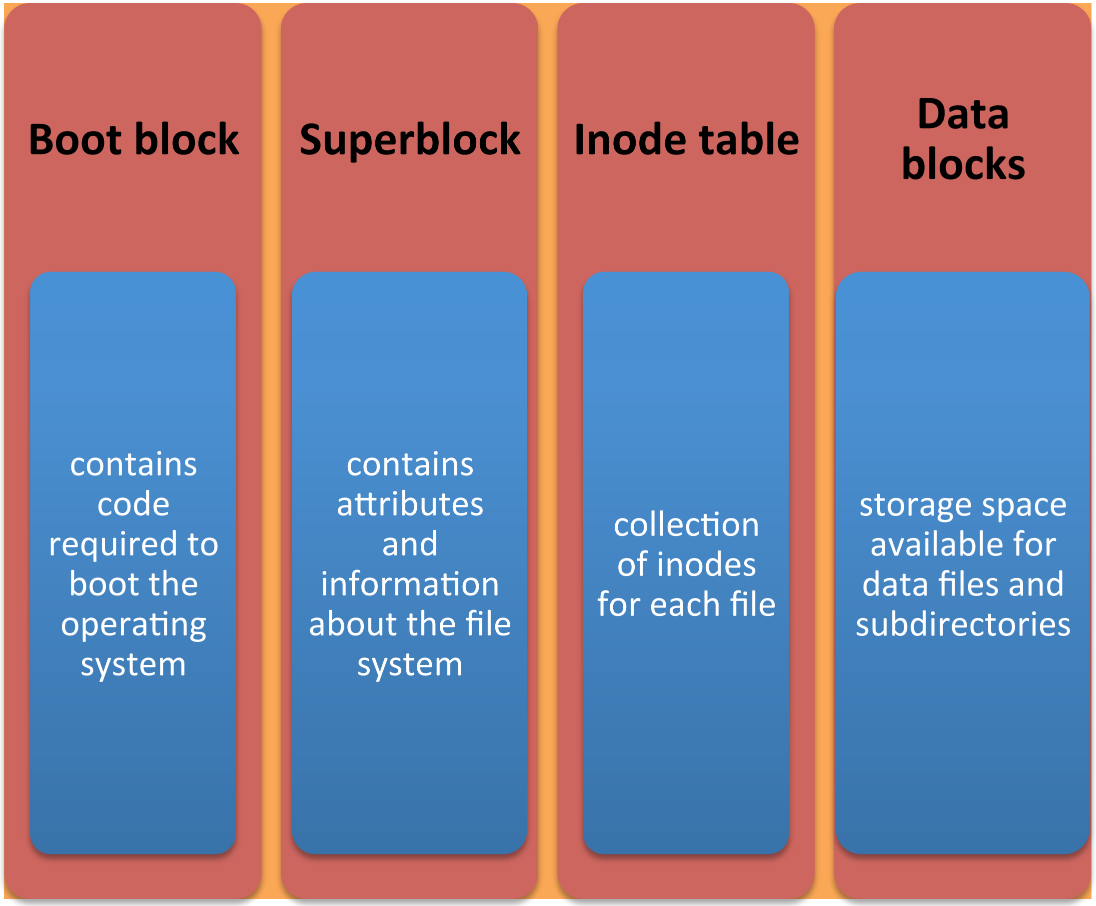
Log Structured File Systems

- Problems
  - Finding an i-node in this log is hard
  - Therefore: maintain “i-node map”:
    - Entries point to i-nodes on disk
    - i-Node map is held in memory and on disk
  - Log structured file system cannot grow infinitely large
    - Write operations write disk segments, a file can be distributed over various segments
    - New data added to the log will reuse stored segments occupied by older versions of file data
    - We therefore need a compaction mechanism that first tries to collect file blocks into contiguous segments, so that freed-up segments on disk can be reused

Volume Management

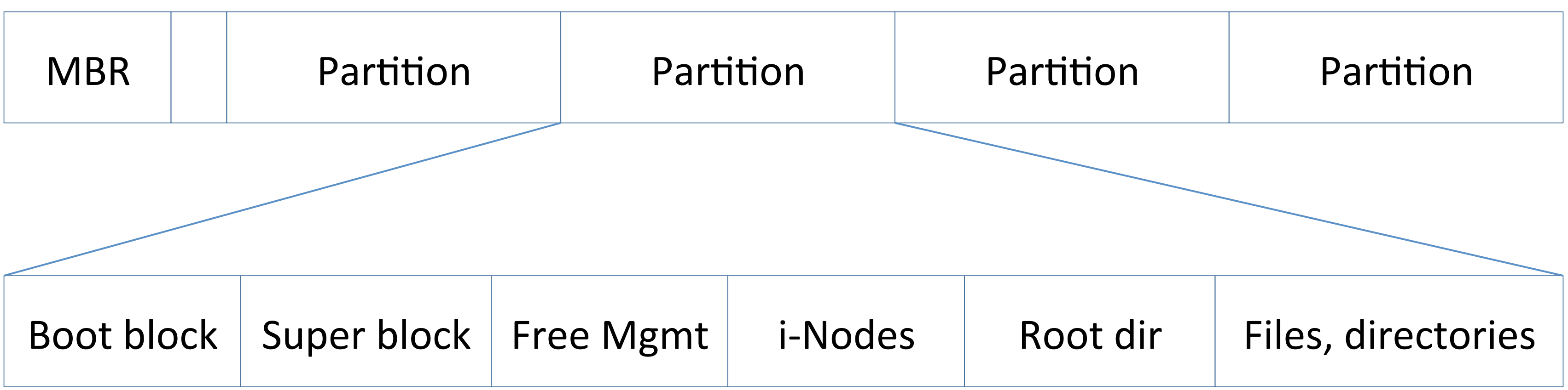
Unix Volume Management

- A UNIX file system resides on a single logical disk or disk partition
- Has a particular layout
  - Boot block
  - Superblock
  - Inode table
  - Data blocks



Unix Disk and File System Layout

- Master Boot Record (MBR)
  - Sector 0 of disk: Contains boot code
  - Partition table
- System start
  - MBR is loaded into memory
  - Program contained in MBR
    - loads the boot block of the active partition, or
    - Provides menu for loading a particular partition



Unix File Management

- Unix distinguishes six types of files
  - Regular or ordinary
    - Contains arbitrary data in zero or more data blocks
  - Directory
    - Contains a list of file names plus pointers to associated indexing information (inodes) pointing to allocated disk blockes
  - Special
    - Contains no data, are not real files, but used to map physical devices to filenames, usual file management functions can be used for read / writes
  - Named pipes
    - Also a kind of file used to create pipes
  - Links
    - Alternative file name for existing file (multiple directory entries for the same file on the disk), data accessible as long as one hard link exists
  - Symbolic links
    - A special file that contains the name of a file it is linked to

File Names

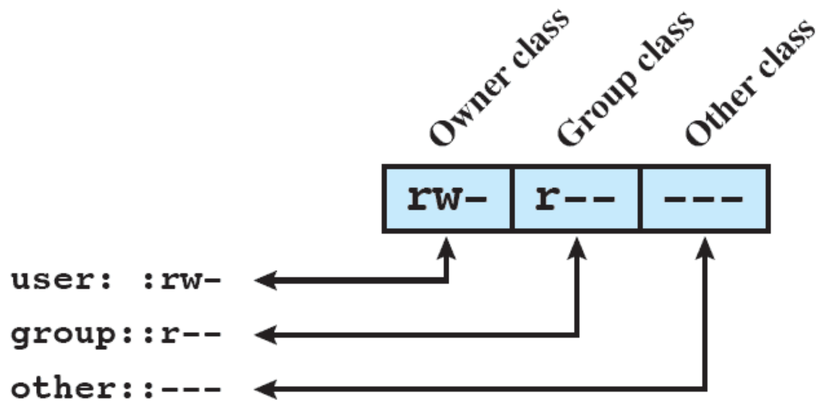
- Operating systems have different conventions
  - Name length: 8 -255 characters
- Case sensitive
  - Unix is case sensitive
  - Windows OS is not case sensitive
- Extensions
  - Windows: extra element that is appended to file name
  - Unix: file names are just strings of characters, may contain ‘.’ and other separator characters to structure a name and give it an “extension”

File Access Control

Unix

- Unique user ID
- Group ID: Each user is a member of a primary group
- Files are owned by a particular user
- Files belong to a group
  - Creator's primary group, or
  - Group of its parent directory, if directory has the "setGID" permission set
- Each file has 12 protection bits
- Owner ID, group ID and protection bits are stored in i-Node

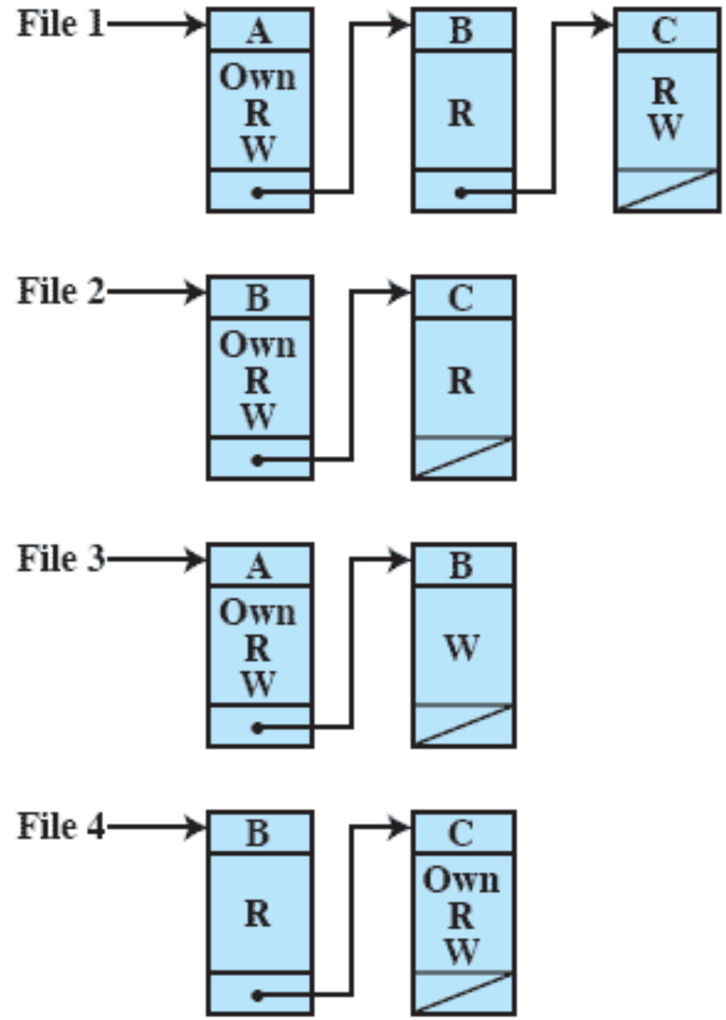
File Access Control



- 12 protection bits
  - Allows the specification of access rights for the owner of the file (user), for members of the primary group of the owner, and for all other users of a system

Access Control Lists

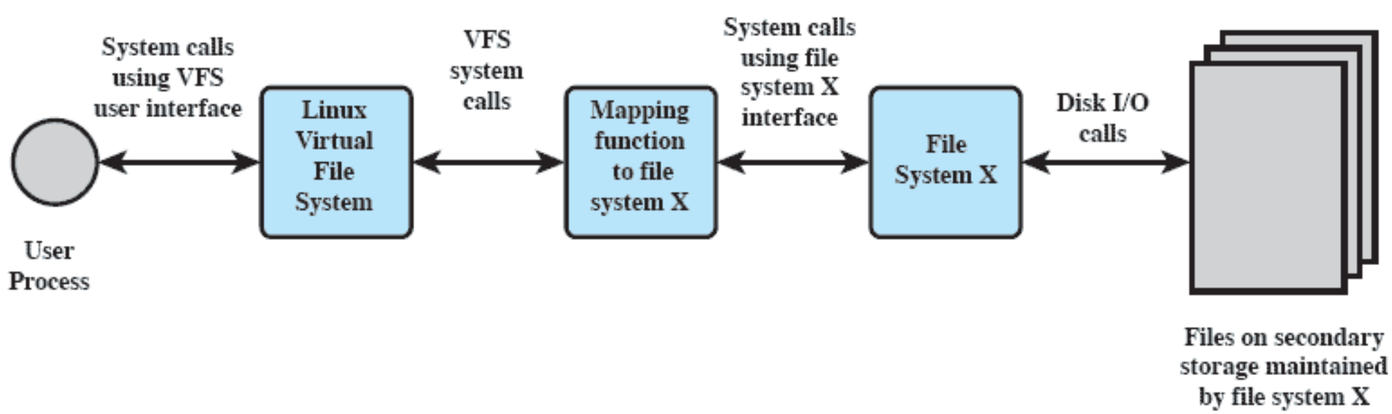
- Allows to assign selected lists of user and group IDs to a file
- Any number of users and groups can be associated with a file, each with three protection bits (read, write, execute)



Linux Virtual File System VFS

Virtual File System

- Linux uses VFS as an abstraction of file systems
  - Acts as a single uniform interface between an actual file system implementation and a user process
  - Defines a common file model that is capable of accommodating general features and behaviour of any conceivable file system implementation
  - Mapping modules translate file access between the VFS and an actual file system



Linux VFS

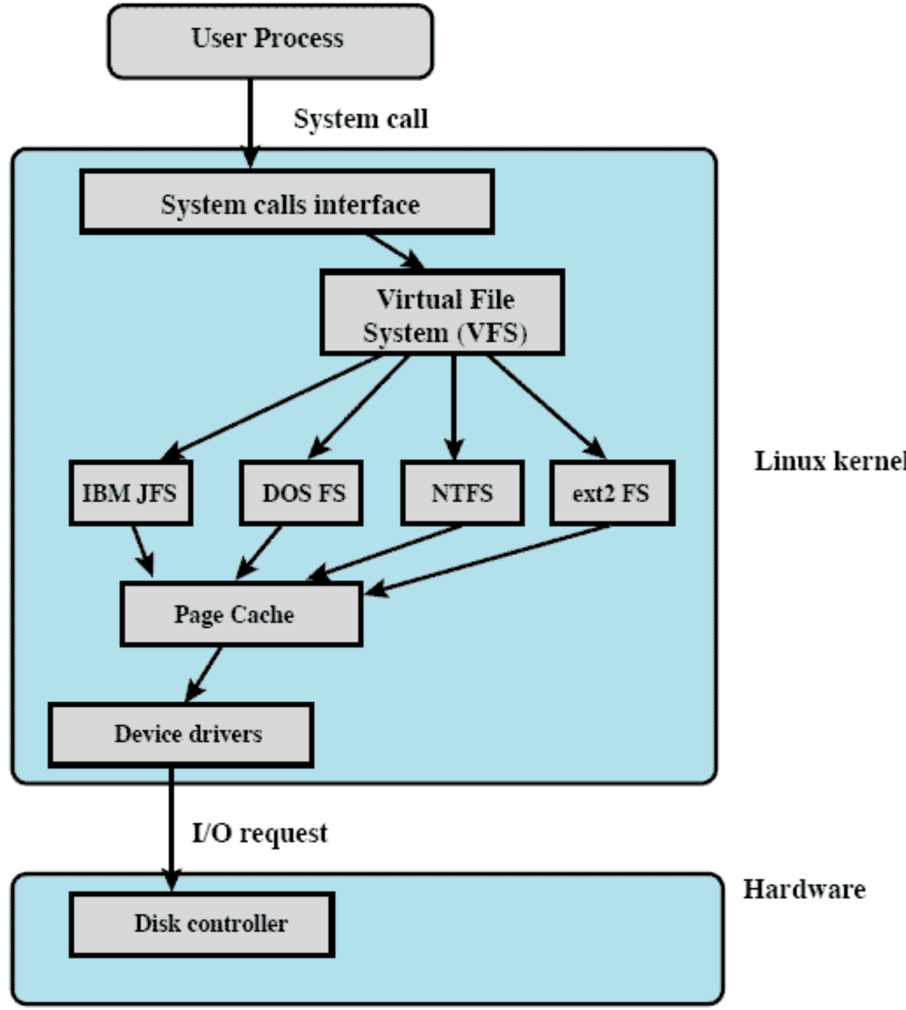


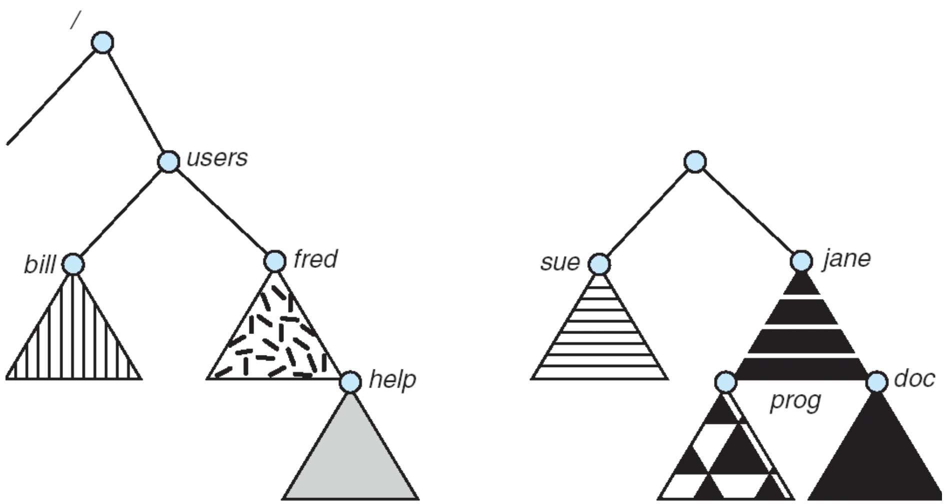
Figure 12.17 Linux Virtual File System Context

Network File System

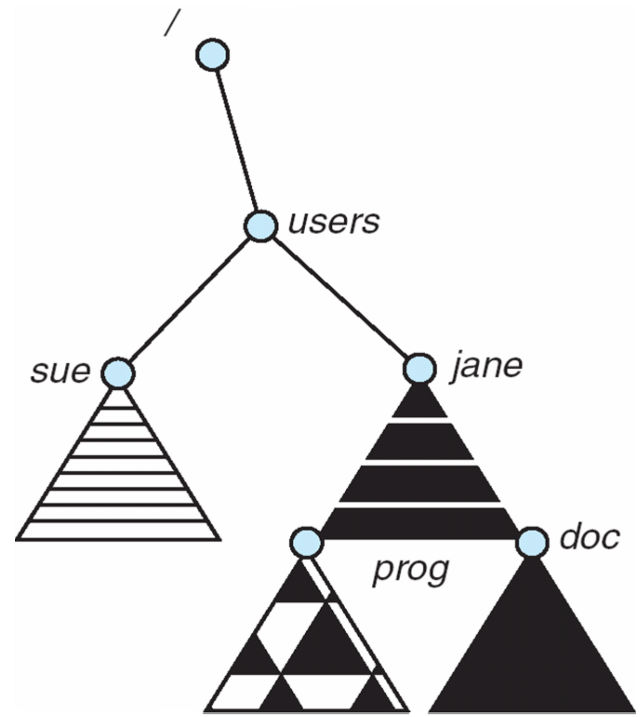
Network File System

- Implementation of a distributed file system for accessing remote files across the network
- Networked computers viewed as independent machines with their independent file systems
- Remote access
  - Mount a remote directory over a local file system directory
  - Files in the remote directory can then be accessed as if they were local
- Based on Remote Procedure Calls (RPC)

Mounting File Systems



(a) (b)

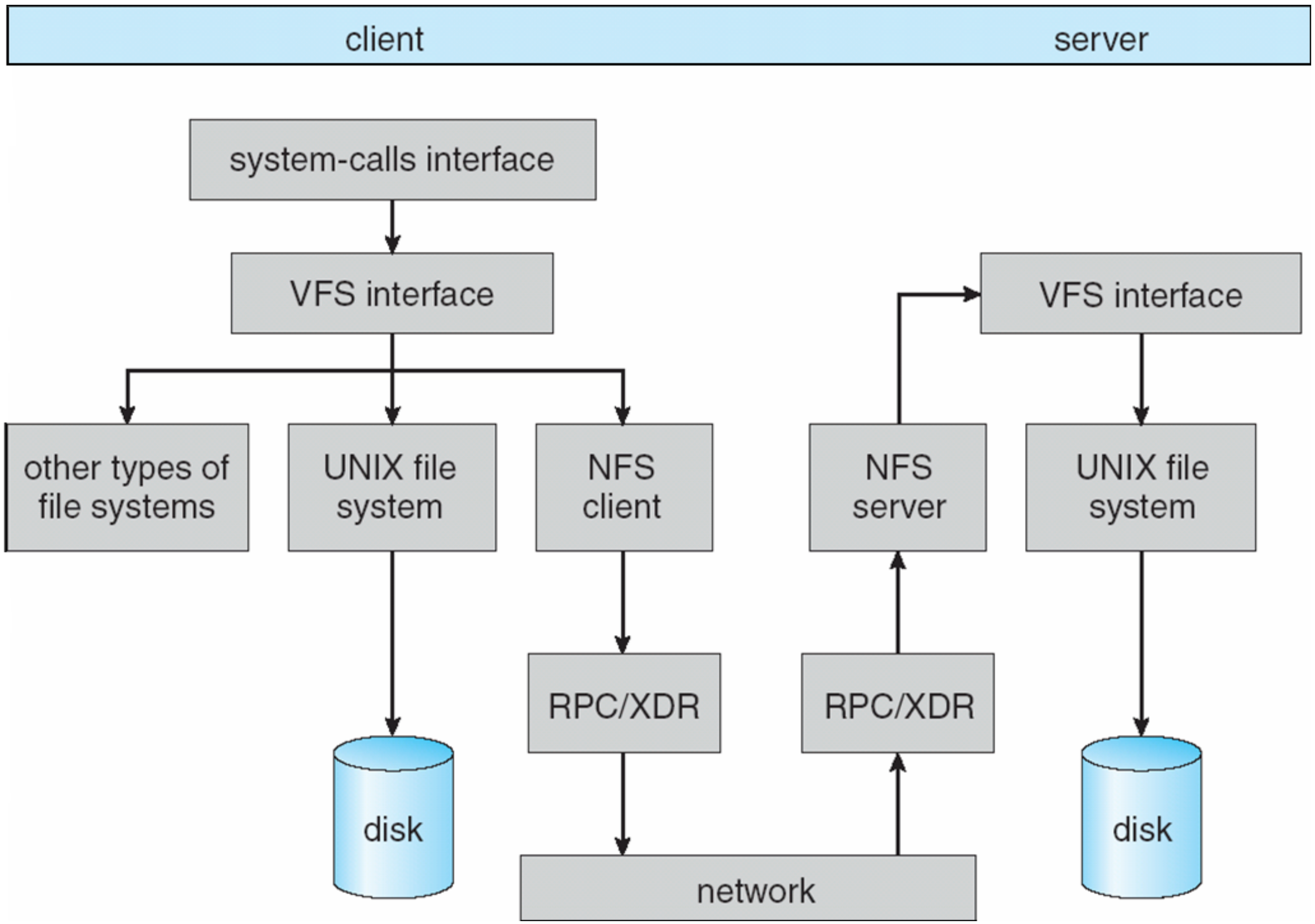


- File Systems have to be mounted at a mount point
  - Is a directory in a directory tree
  - Will be overlaid by mounted directory structure
  - Directory /mnt is the traditional directory where mount points can be collected

Three Major Layers of NFS

- Unix file system interface (open, read, write, close, file descriptor datastructures)
- Virtual File system (VFS) layer
  - Distinguishes local files from remote files and files according to their file system types
    - VFS activates file system specific operations to handle local requests according to file system type
    - Calls the NFS protocol procedures for remote requests
- NFS service layer
  - Implements the NFS protocol

NFS


client
server
system-calls interface
VFS interface
other types of file systems
UNIX file system
NFS client
disk
RPC/XDR
network
VFS interface
NFS server
UNIX file system
RPC/XDR
disk